CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 11

1. (20 pts.) **Evacuation Dynamic Program.**

1. The subproblems are the number of paths to every intermediate node. To implement the solution, you would make an array A that tracks the number of paths to every node between the start (index 0) and end (final index) in topological order.
2. The base case is the number of paths to the starting node, which is 1. $A[0] = 1$.
3. $A[j] = \sum_{(i,j)\in E} A[i] \qquad \forall j \geq 1$

2. (25 pts.) **Weighted Set Cover.** As in the unweighted case, we will use a greedy algorithm:

```
while(some element of (B) is not covered)
{
    Pick the set (S_i) with the largest ratio
    ((new elements covered by (S_i)) /w_i).
}
```

Now we will prove that if there is a solution of cost k, then the above greedy algorithm will find a solution with cost at most $k \log_e n$. Our proof is similar to the proof in the unweighted case in Section 5.4 of the textbook.

After t iterations of the algorithm, let n_t be the number of elements still not covered, and let k_t be the total weight of the sets the algorithm chose, so $n_0 = n$ and $k_0 = 0$. Since the remaining n_t elements are covered by a collection of sets with cost k, there must be some set S_i such that S_i covers at least $w_i n_t/k$ new elements. (This is easiest to see by contradiction: if every set S_i covers less than $w_i n_t/k$ elements, then any collection with total weight k will cover less than kn_t/k elements.) Because some set with such a desirable ratio exists, our greedy algorithm can pick that set to ensure that it is always picking a set with ratio at least $w_i n_t/k$. Therefore the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t(k_{t+1} - k_t)}{k} = n_t(1 - (k_{t+1} - k_t)/k).$$

Now, we apply the fact that for any x, $1 - x \leq e^{-x}$, with equality iff $x = 0$:

$$n_{t+1} < n_t e^{-(k_{t+1} - k_t)/k}.$$

By induction, we find that for $t > 0$, $n_t < n_0 e^{-k_t/k}$. If we choose the smallest t such that $k_t \geq k \log_e n$, then, n_t is strictly less than 1, which means no elements remain to be covered after t steps. Note that we will never add a set of weight more than k, due to the relationship discussed in paragraph 2. There exists some set that covers at least $w_i n_t/k$ elements, which implies that set's ratio is at least n_t/k. Therefore, the only time we can choose a set with weight k is if it covers all remaining elements, and it is not possible to choose a set with weight more than k. It follows that $k_t < k \log_e n + k = O(k \log n)$.

3. (25 pts.) **Exponential Greedy Solutions.** To create exponentially many possible solutions, we could construct an instance that encounters a tie between a constant number of elements at every step and runs for $O(n)$ steps. However, the easier approach is to construct an instance that encounters a single tie between an

exponential number of sets. Let the set of items $U = \{1, 2, \ldots, n\}$. We first construct a set $S_1 = \{1, 2, \ldots, n-1\}$. We want the greedy algorithm to pick S_1 first. Then, we want any other set to be able to cover item n in the second step of the algorithm, creating a tie between all remaining sets. Now we construct an exponential number of other subsets S_i that all cover n, while making sure none of them become a better first option than S_1. To keep things simple, let's accomplish this by saying none of the S_i include items 1 or 2, but all of them include item n. This leaves $n-3$ items, and we will construct one set for every permutation of including and excluding those items, for $2^{n-3} = O(2^n)$ sets total. At the second step of the greedy algorithm, there will be a tie between all $O(2^n)$ sets, creating $O(2^n)$ possible greedy solutions.

4. (25 pts.) **Longest Common Substring.** *Algorithm:* For $0 \leq i \leq n$ and $0 \leq j \leq m$, define $L(i, j)$ to be the length of the longest common postfix of $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$. When $i = 0$, it indicates $x_1 x_2 \ldots x_i$ is an empty string. Similarly, when $j = 0$, it indicates $y_1 y_2 \ldots y_j$ is an empty string. The recursion is:

$$L(i, j) = \begin{cases} L(i-1, j-1) + 1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

The initialization is $L(0, 0) = 0$. Then, for all $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$L(i, 0) = 0$$
$$L(0, j) = 0$$

The length of the longest common substring of x and y is the maximum of $L(i, j)$ over all $1 \leq i \leq n$ and $1 \leq j \leq m$. We can find the maximum by either tracking the maximum when solving all $L(i, j)$ or going through all those values when they are calculated.

Correctness and Running Time: The initialization is clearly correct as there is no postfix for an empty string. The heuristic to solve this problem is that the longest common substring of x and y must be the longest common postfix among all possible substrings of x and all possible substrings of y. However, we don't need to go through all substrings of x and y since we only care about the postfixes i.e. the ending parts. Thus, going through all possible combinations of $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$ is enough.
This implies we have $O(mn)$ subproblems regarding $L(i, j)$. Since each takes constant time to evaluate if we have the previous results (which will be the case as we use dynamic programming), the time to solve all subproblems is $O(mn)$. Either method of finding the maximum keeps the overall running time at $O(mn)$.

Alternative Solution:
Algorithm: For $1 \leq i \leq (n+1)$ and $1 \leq j \leq (m+1)$, define $L(i, j)$ to be the length of the longest common prefix of $x_i x_{i+1} \ldots x_n$ and $y_j y_{j+1} \ldots y_m$. When $i = n+1$, it indicates $x_i x_{i+1} \ldots x_n$ is an empty string. Similarly, when $j = m+1$, it indicates $y_j y_{j+1} \ldots y_m$ is an empty string. The recursion is:

$$L(i, j) = \begin{cases} L(i+1, j+1) + 1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

The initialization is $L(n+1, m+1) = 0$. Then, for all $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$L(i, m+1) = 0$$
$$L(n+1, j) = 0$$

The length of the longest common substring of x and y is the maximum of $L(i, j)$ over all $1 \leq i \leq n$ (starting from n going back to 1) and $1 \leq j \leq m$ (starting from m going back to 1). We can find the maximum by either tracking the maximum when solving all $L(i, j)$ or going through all those values when they are calculated.

Correctness and Running Time: The initialization is clearly correct as there is no prefix for an empty string. The heuristic to solve this problem is that the longest common substring of x and y must be the longest common prefix among all possible substrings of x and all possible substrings of y. However, we don't need to go through all substrings of x and y since we only care about the prefixes i.e. the beginning parts. Thus, going through all possible combinations of $x_i x_{i+1} \dots x_n$ and $y_j y_{j+1} \dots y_m$ is enough.
This implies we have $O(mn)$ subproblems regarding $L(i, j)$. Since each takes constant time to evaluate if we have the previous results (which will be the case as we use dynamic programming), the time to solve all subproblems is $O(mn)$. Either method of finding the maximum keeps the overall running time at $O(mn)$.

Rubric:

Problem 1, 25 pts

Directly from solutions to homework 6.

(a) 8 pts

(b) 8 pts

(c) 9 pts

Problem 2, 25 pts

- 10 pts for giving the algorithm including:

```
while(some element of (B) is not covered
{
    Pick the set (S_i) with the largest ratio
    ((new elements covered by (S_i)) /w_i).
}
```

- 15 pts for proving the algorithm

Problem 3, 25 pts

- 5 pts for identifying at least one case where ties lead to an exponential number of solutions
- 20 pts for the set cover instance (might stack many ties instead of one huge tie).

Problem 4, 25 pts

- 5 points: subproblems
- 5 points: recurrence relation
- 2 points: initialization
- 2 points: retrieving final answer from computed DP
- 5 points: proof of correctness
- 5 points: running time analysis